Tasks: Create Your Own Music Recommendation System Using Machine Learning

Using the Spotify Web API, we have extracted musical features for a great number of playlists. Each playlist belongs to one of 22 categories: Pop, Hip-Hop, Electronic/Dance, R&B, Rock, Indie, Metal, Country, 90s, 80s, 70s, 60s, Soul Jazz, Latin, Folk & America, Classical, Reggae, Blues, Funk, Punk og Romance/Love songs. Each of the categories consist of approximately 250-400 songs, intended to be a representative selection.

The Spotify Web API returns 13 features to describe the musical aspects of a song. Below, you can see an example of what the API returns for a particular song, in addition to a table describing each feature.


```
{
  "danceability" : 0.735,
  "energy" : 0.578,
  "key" : 5,
  "loudness" : -11.840,
  "mode" : 0,
  "speechiness" : 0.0461,
  "acousticness" : 0.514,
  "instrumentalness" : 0.0902,
  "liveness" : 0.159,
  "valence" : 0.624,
  "tempo" : 98.002,
  "type" : "audio_features",
  "id" : "06AKEBrKUckW0KREUWRnvT",
  "uri" : "spotify:track:06AKEBrKUckW0KREUWRnvT",
  "track_href" : "https://api.spotify.com/v1/tracks/06AKEBrKUckW0KREUWRnvT",
  "analysis_url" : "https://api.spotify.com/v1/audio-analysis/06AKEBrKUckW0KREUWRnvT",
  "duration_ms" : 255349,
  "time_signature" : 4
}
```


Feature	Type	Description
acousticness	float	float A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
analysis_url	string	An HTTP URL to access the full audio analysis of this track. An access token is required to access this data.
danceability	float	Danceability describes how suitable a track is for dancing based on a combination of musical elements

		including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
duration_ms	int	The duration of the track in milliseconds.
energy	float	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
id	string	The Spotify ID for the track.
instrumentalness	float	Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.
key	int	The key the track is in. Integers map to pitches using standard Pitch Class notation . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on.
liveness	float	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
loudness	float	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typical range between -60 and 0 db.
mode	int	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
speechiness	float	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording

		(e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
tempo	float	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.
time_signature	int	An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).
track_href	string	A link to the Web API endpoint providing full details of the track.
type	string	The object type: “audio_features”
uri	string	The Spotify URI for the track.
valence	float	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

Assignment 0: Statistics (15 min)

Let’s start slowly. In this preliminary task you will construct a dataset and take a look at the features of the songs you like. Optimally this dataset would be handpicked, where every song represents a song you like, but this process would be very time-consuming and not viable for this workshop, so instead you will pick some of the playlists we have collected for you.

1. Before anything else, you have to open the file located at *dataset/dataset_config.py* and pick 10-15 playlists. Each line in the list stored as *likeable_songs* represents a playlist. Un-comment the playlists you believe represent your musical taste. If you want to be certain, you can check the track list in each genre folder, but do not spend too much time :)
2. Run assignment0/run_statistics.*py* to output the standard deviation and the average values for a subset of features.

3. How is the variation in your dataset? Do you mostly have very similar songs or is there a large degree of variation? How is the standard deviation? A low value indicates low variation in your musical taste :) Note that your funness-score is printed out in the console. How fun and energetic is your dataset? Of course this is really a matter of subjective opinion, but nevertheless, we calculate you funness-score from a subset of the features. The formula is as follows:
   "**funness" = loudness + tempo + (energy*100) + (danceability*100)**

   Compare with your new buddies around you to see who's most fun! :)

Assignment 1: Clustering (45 min)

Let us start looking at some real machine learning algorithms. In this task we're going to cluster the dataset, and see some natural groups formed based on the similarity of songs.

1. Run *run_assignment1.py* to output figures illustrating the clusters being formed by the three different algorithms. To view the figure for the next algorithm, close the window (X) for the current figure. To only run one algorithm at a time you can comment out the run methods at the bottom of the file.

   In the files *kmeans.py, agglomerative.py* and *dbscan.py* you can change the parameters of the respective algorithms. E.g., try to change the number of clusters to construct by the k-means algorithm :) Study the graphs and see if you see anything interesting. From your perspective, do the clusters make sense? Use the mouse pointer and hover over the data points to display song information.
2. If you have not already done so, hand-pick the features you would like to use for clustering songs. Try different selections and see if this affects your clusters in any way. Does any of the features seem to make more impact that the others. Please do compare and discuss with your comrades.

Assignment 2: Classification (45 min)

In this task you will train an SVM and a neural network, which will be able to predict how much you will like a given song. This you will do by training on two sets of data: one set of songs you like, and one set of songs you dislike.

2A: Dataset

If you have not already done so, head back to *dataset/dataset_config.py* and uncomment 4-6 categories representing genres you do do not like. Do this is in the

list stored in *annoying_songs*. The greater the dataset, the better, but we do not want to spend too much time setting up the dataset in this workshop. The playlists contain different amount of songs. Try to end up with approximately 1500-2000 songs in the dataset, and a 50/50 distribution with likeable and annoying songs. When you execute *assignment2*/*run_classification.py,* the number of songs in each set will be printed.

2B: SVM

We are going for an easy start by setting up an out-of-the-box SVM, which is only a few lines of code using the scikit-learn framework. You will get to play around a little more with the neural network.

1. Head to *assignment2/run_classification.py* and uncomment train_svm() at the bottom of the file.
2. Instantiate an SVM object with this line: *svm.SVC(probability=True),* and store in the *clf* variable.
3. Train the SVM by calling *.fit()* on the SVM object and pass the training set *(x_train)* and labels *(y_train)* as arguments.
4. Store the prediction results on the validation set in *val_score* by calling *.score()* and passing the validation set and labels as arguments.
5. Run the file and you will get prediction accuracies on the training set and validation set. The model is automatically saved for the SVM.

Music recommendation

Result-wise, this is where we get to the interesting part. Now we are going to use the model to output recommendations. For this, we have downloaded a large playlist containing around 8000 songs from different genres. In this context, we will use this to represent the “all the songs in the world” (we know it’s not). Unlike Spotify, we do not have simple access to the actual library of all songs in the world. We are going to run this playlist through our trained model. The model will then give us a probability score describing how likely it is that we will like the each song, based on patterns it captures.

- Uncomment the *music_recommendation_svm()* method at the end of the file and run. The output will show 30 songs the model thinks you will like. For fun, it also shows you 30 songs you will definitely dislike with passion.
- Do the results seem to be right based on the songs you recognize (or don’t recognize)?

2C: Neural Networks

1. Head to *assignment2/run_classification.py* and uncomment train_nn() at the bottom of the file, while commenting out *train_svm().*

2. Try out different parameter values and architectures for the neural network, inside the *train_nn()* method and run to train the model. For example, you can make the model more complex by adding hidden layers. This you can do by adding Dense() layers to the model. You can also choose a larger number of neurons in the hidden layers (Dense layer).

   Study the outputted graphs while trying different values. They show the amount of loss (error) and accuracy for the training set and validation set during training. Do you see any changes, for the better or worse, as a result of your decisions?
   - The number of hidden neurons to choose is not straightforward, but there are some rules of thumb. Usually it should be somewhere between the size of the input layer and the size of the output layer.

     The initial setup only has one neuron in a single hidden layer. Try increasing this number. Does your model learn more?

3. When you are happy with your model, you can save it by changing the argument for the save_model parameter in *train_nn()* from *False* to *True.* Remember to change this at the bottom of the file where the method is called and not the default parameter in the method definition.
4. As with the svm, get recommendations by uncommenting the method called *music_recommendation_nn*(). It works in a similar fashion.

Optional neural network tasks:

1. **Dropout**
   - By adding more layers or neurons, your model expands its capacity of capturing patterns in the training set, but this may lead to overfitting. You can counteract this by adding a Dropout layer after your hidden layer. This will omit a certain percentage of the neurons in your hidden layer in every iteration. Practically speaking, this simulates the use of several neural networks for training, and thus reduces the chance of overfitting.

     Use dropout by adding an instance of *Dropout(rate)* to you model, exactly as you have done with Dense layers earlier.

2. **Early stopping**
   - Depending on the configuration and your architecture, you may see from your training graphs that if you train for many epochs the performance on the validation set stops at a certain point. If this is the model architecture you are going for, then it would be a good idea to

stop training at the point where performance stops to improve. Do this by using early stopping, which monitors a certain value and stops training when that value stops to improve.

Use early stopping by adding an instance of *EarlyStopping(monitor="val_loss", patience=2)* in the list of callbacks for your model. This will monitor the validation loss and stop training if it does not improve over two epochs, thus saving the best possible model following this configuration.